Practice Midterm 4 Solutions

This page contains solutions to Practice Midterm 4.

Question 1: Moira's Teashop of Wonderment and Whimsy

This task was mostly about implementing a given specification and demonstrating that you know how to correctly use ADTs, including a nested collection type. One of the most common errors we saw was the application of set functions that modified a pass-by-reference parameter in a way that left it in a ruined state upon returning from the `teaFromSource()` function. One way around that would be to create a copy of one of the sets before applying any of the set functions.

```
Set<string> teaFromSource(infoT& target, infoT& source) {
    Set<string> intersection = target.favorites * source.favorites;
    if (intersection.size() >= target.favorites.size() / 2.0) {
        return source.favorites - target.favorites;
    }
    return {};
}

Map<string, Set<string>> getRecommendations(Vector<infoT>& allTheTea) {
    Map<string, Set<string>> map;
    for (int i = 0; i < allTheTea.size(); i++) {
        for (int j = 0; j < allTheTea.size(); j++) {
            if (i != j) {
                map[allTheTea[i].name] += teaFromSource(allTheTea[i],
allTheTea[j]);
            }
        }
    }
    return map;
}
```

Question 2: Runtime Analysis (Big-O)

`indexIsValid()` (pass-by-value)

This one is O(n) (best- and worst-case) because the vector is being passed by value – which means the first thing that happens when we call this function is that we spend O(n) time creating a copy of the vector.

- Best Case: O(n)
- Worst Case: O(n)

`indexIsValid()` (pass-by-reference)

- Best Case: O(1)
- Worst Case: O(1)

`countAndDeplete()`

Recall that removing from index 0 from a vector is an O(n) operation because all the remaining

- Deleting all elements and shifting left one-by-one.
  - With `v.remove(0)` in loop: O(n^2)
  - With `v.remove(v.size() - 1)` in loop: O(n)

`contains()`

- Best Case: O(1)
- Worst Case: O(log n)

Question 3: Recursion (Postfix)

This recursive problem had a similar flavor to the `countFavorableOrderings()` problem from Assignment 3. A strongly related problem on postfix expressions was also presented in the Exam Prep section of the Lecture 5 notes.

Here's some advice from one of the senior SLs who helped grade this question, in case you encounter a question like this on your midterm this quarter:

- Read the question thoroughly. A lot of information is actually given in the instructions, and it's always a bummer to take points for something that was explicitly mentioned (ex. not calculating the postfix expression, not using ADTs, not modifying the vector)! Make sure you're reading the entire premise and tips on implementation (I love underlining key parts so I don't forget them!) so that you're not making little mistakes.
- Trust and understand the recursive step. While there are a lot of ways to approach a recursive problem, it's important to keep track of the one approach you're going to take. Always think: what is my base case? Is this recursive step going to take me closer to my base case? What do I actually want to accomplish?
- Be mindful with helper function parameters. Similar to the above tip, we should only keep the parameters that are actually necessary, as having more can be confusing and get away from the real recursive process that we want to occur. Sometimes, less parameters is more!

Here are three solutions to the problem, all of which are slight twists on one another:

Solution 1

This one keeps track of the number of operators and numeric values we have seen so far as we process the expression. It relies on the observation that the number of values we have seen must always exceed the number of operators by at least one as we process the expression, and in the end, there must be exactly one more value than number of operators.

```
bool isValidHelper(Vector<string>& expr, int numOperators, int numVals) {
    if (expr.isEmpty()) {
        return numVals - numOperators == 1;
    }

    if (numOperators > 0 && numVals - 1 < numOperators) {
        return false;
    }

    string thisOne = expr.remove(0);

    if (isOperator(thisOne)) {
        return isValidHelper(expr, numOperators + 1, numVals);
    }

    if (isNum(thisOne)) {
        return isValidHelper(expr, numOperators, numVals + 1);
    }

    return false;
}

bool isValid(Vector<string> expr) {
    return isValidHelper(expr, 0, 0);
}
```

Solution 2

Similar to the solution above, this one only tracks the number of unresolved numeric values we have seen so far as we process the expression. Applying an operator would take two numeric values to its left and replace them with a single value (the result of applying that operator), and so the number of values gets decremented by one.

```
bool isValidHelper(Vector<string>& expr, int numVals) {
    if (numVals < 0) {
        return false;
    } else if (expr.size() == 0) {
        return numVals == 1;
    }

    string thisOne = expr.remove(0);

    if (isNum(thisOne)) {
        return isValidHelper(expr, numVals + 1);
    }
    if (isOperator(thisOne)) {
        if (numVals < 2) {
            return false;
        }

        return isValidHelper(expr, numVals - 1);
    }

    return false;
}

bool isValid(Vector<string> expr) {
    return isValidHelper(expr, 0);
}
```

Solution 3

This is another valid solution which shows an alternative approach to the base cases and an alternative if-else structure.

```
bool isValidHelper(Vector<string>& expr, int numVals) {
    if (expr.size() == 0) {
        return false;
    } else if (expr.size() == 1) {
        return (numVals == 0 && isNum(expr[0])) || (numVals == 2 &&
isOperator(expr[0]));
    } else if (isOperator(expr[0])) {
        if (numVals < 2) {
            return false;
        } else {
            expr.remove(0);
            return isValidHelper(expr, numVals - 1);
        }
    } else if (isNum(expr[0])) {
        expr.remove(0);
        return isValidHelper(expr, numVals + 1);
    } else {
        return false;
    }
}

bool isValid(Vector<string> expr) {
    return isValidHelper(expr, 0);
}
```

Question 4: Recursive Backtracking (sumWords)

There are several possible approaches to this problem. Below are two of the most straightforward ones.

Solution 1

This solution adds our ASCII values until we meet or exceed the target.

```
int sumWordsHelper(string soFar, int target, int sumSoFar, Lexicon& lex) {
    if (!lex.containsPrefix(soFar)) {
        return 0;
    }
    if (sumSoFar > target) {
        return 0;
    }
    if (sumSoFar == target) {
        if (lex.contains(soFar)) {
            cout << soFar << endl;
            return 1;
        }
        return 0;
    }

    int total = 0;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        total += sumWordsHelper(soFar + ch, target, sumSoFar + ch, lex);
    }

    return total;
}

int sumWords(int target, Lexicon& lex) {
    return sumWordsHelper("", target, 0, lex);
}
```

Solution 2

This solution is very similar to the one above, but it subtracts ASCII values directly from the target directly until we meet or dip below zero.

```
int sumWordsHelper(string soFar, int target, Lexicon &lex) {
    if (target < 0) {
        return 0;
    }
    if (target == 0) {
        if (lex.contains(soFar)) {
            cout << soFar << endl;
            return 1;
        }
        return 0;
    }
    if (!lex.containsPrefix(soFar)) {
        return 0;
    }

    int total = 0;
    for (char ch = 'a'; ch <= 'z'; ch++) {
        total += sumWordsHelper(soFar + ch, target - ch, lex);
    }

    return total;
}

int sumWords(int target, Lexicon& lex) {
    return sumWordsHelper("", target, lex);
}
```

Website Programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-Apr-21